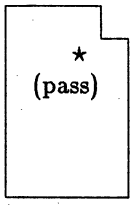
THE PORTABLE STANDARD LISP USERS MANUAL

PART 1: LANGUAGE SPECIFICATION

MACINTOSH^{tm} VERSION 1.0



Utah Portable Artificial Intelligence Support Systems Project
Computer Science Department
University of Utah
Salt Lake City, Utah 84112
Version 3.2: 12 April 1984
Modified for Macintosh^{tm} PSL - April 25, 1985

ABSTRACT

This manual describes the primitive data structures, facilities and functions present in the Portable Standard Lisp (PSL) system. It describes the implementation details and functions of interest to a PSL programmer. Except for a small number of hand-coded routines for I/O and efficient function calling, PSL is written entirely in itself, using a machine-oriented mode of PSL, called SYSLisp, to perform word, byte, and efficient integer and string operations. PSL is compiled by an enhanced version of the Portable Lisp Compiler, and currently runs on the DEC-20, VAX, and MC68000.

Work supported in part by the Burroughs Corporation, the Hewlett Packard Company, the International Business Machines Corporation, the National Science Foundation Under Grant Numbers MCS80-07034, MCS81-21750 and MCS82-04247 and the Defense Advanced Research Projects Agency under contract number DAAK11-84-K-0017.

Table of Contents

1 PREFACE

This Portable LISP implementation would not have been started without the effort and inspiration of the original STANDARD LISP reporters (A. C. Hearn, J. Marti, M. L. Griss and C. Griss) and the many people who gave freely of their advice (often unsolicited!). We especially appreciate the comments of A. Norman, M. Rothstein, H. Stoyan and T. Ager.

It would not have been completed without the efforts of the many people who have worked arduously on SYSLISP and PSL at various levels: Eric Benson, Will Galway, Ellen Gibson, Martin Griss, Bob Kessler, Steve Lowder, Chip Maguire, Beryl Morrison, Don Morrison, Bobbie Othmer, Bob Pendleton, John Peterson, and John W. Peterson.

We are also grateful for the many comments and significant contributions by the LISP users at the Hewlett-Packard Computer Research Center in Palo Alto.

We would also like to thank Bill Croft and the Sumex Macintosh^{tm} project. These tools made developing the Mac interface, much easier and allowed all of the development to proceed on our Vax.

This document has been worked on by most members of the current Utah Symbolic Computation Group. The primary editorial function has been in the hands of B. Morrison, M. L. Griss, B. Othmer, and W. Galway; major sections have been contributed by E. Benson, W. Galway, and D. Morrison. There have also been significant contributions to the manual from Hewlett-Packard.

This Macintosh^{tm} version has been created by removing those functions that are not available within the Macintosh^{tm} PSL. This editing has been performed by Robert R. Kessler, with editorial comments by Gerald Q. Maguire Jr.

We have reorganized the manual for this version, following the Common Lisp idea of having four parts for language definition, utilities, system-dependent information, and implementation information. Most of this reorganization was done at Hewlett-Packard.

This is a preliminary version of the manual, and so may suffer from a number of errors and omissions. Please let us know of problems you may detect.

Report bugs, errors and mis-features by sending MAIL to PSL-BUGS@Utah-20.

2 INTRODUCTION

2.1 Opening Remarks

This document describes PSL (PORTABLE STANDARD LISP[1]), a portable, "modern" LISP developed at the University of Utah for a variety of machines. PSL is upward-compatible with STANDARD LISP [Marti 79]. In most cases, STANDARD LISP did not commit itself to specific implementation details (since it was to be compatible with a portion of "most" LISPs). PSL is more specific and provides many more functions than described in that report.

The goals of PSL include:

- Providing implementation tools for LISP that can be used to implement a variety of LISP-like systems, including mini-LISPs embedded in other language systems (such as existing PASCAL or ADA applications).
- Effectively supporting the REDUCE algebra system on a number of machines, and providing algebra modules extracted from (or modeled upon) REDUCE to be included in applications such as CAI and CAGD.
- Providing a uniform, modern LISP programming environment on all of the machines that we use (DEC-20, VAX, and 68000 based personal machines)– of the power of FRANZ LISP, UCI LISP or MACLISP.
- Studying the utility of a LISP-based systems language for other applications (such as CAGD or VLSI design) in which SYSLISP code provides efficiency comparable to that of C or BCPL, yet enjoys the interactive program development and debugging environment of LISP.

2.2 Scope of the Manual

This manual is intended to describe the syntax, semantics, and implementation of PSL. While we have attempted to make it comprehensive, it is not intended for use as a primer. Some prior exposure to LISP will prove very helpful. A selection of LISP primers is listed in the bibliography in Chapter 16; see for example [Allen 79, Charniak 80, Weissman 67, Winston 841].

The PSL documentation is divided into four parts following the Common LISP practice. Part 1, the "white pages" (this document), is a language specification. Part 2, the "yellow pages", is a program library document. Part 3, the "red pages", is implementation-dependent documentation. You may want to consider reading Part 3 first.

[1] "LSP" backwards!

2.2.1 Typographic Conventions within the Manual

A large proportion of this manual is devoted to descriptions of the functions that make up PSL. Each function is provided with a prototypical header line. Each argument is given a name and followed by its allowed type. If an argument type is not commonly used, it may be a specific set enclosed in brackets {...}. For example, this header shows that PutD (which defines other functions) takes three arguments:

(PutD FNAME:id TYPE:ftype BODY:{lambda,code-pointer}): id	*expr*

1. FNAME, which is an id (identifier).
2. TYPE, which is the "function type" of the function being defined.
3. BODY, which is a lambda expression or a code-pointer.

and returns FNAME, the name of the function being defined. Some functions are compiled open; these have a note saying "open-compiled" next to the function type.

Some functions accept an arbitrary number of arguments. The header for these functions shows a single argument enclosed in square brackets– indicating that zero or more occurrences of that argument are allowed. For example:

(And [U:form]): extra-boolean	*fexpr*

And is a function which accepts zero or more arguments each of which may be any form.

In some cases, LISP code is given in the function documentation as the function's definition. As far as possible, the code is extracted from the the current PSL sources (perhaps converted from one syntax to the other); however, this code is not always necessarily used in PSL, and may be given only to clarify the semantics of the function. Please check carefully if you depend on the exact definition.

Some features of PSL are anticipated but not yet fully implemented. When these are documented in this manual they are indicated with the words: [not implemented yet].

2.2.2 The Organization of the Manual

Here is a brief overview of the following chapters:

Chapter 2 describes the data types used in PSL. It includes functions useful for testing equality and for changing data types, and predicates useful with data types.

The next seven chapters describe in detail the basic functions provided by PSL.

Chapters 3, 4, 5, and 6 describe functions for manipulating the basic data structures of LISP: numbers, ids, lists, and strings and vectors. As virtually every LISP program uses integers, identifiers, and lists extensively, these three chapters (3, 4 and 5) should be included in an overview. As vectors and strings are used less extensively, Chapter 6 may be skipped on a first reading.

Chapter 7 and, to some extent, Chapter 2 describe the basic functions used to drive a computation. The reader wanting an overview of PSL should certainly read these two.

Chapter 8 describes functions useful in function definition and the idea of variable binding. The novice LISP user should definitely read this information before proceeding to the rest of the manual.

Chapter 9 describes functions associated with the interpreter. It includes functions having to do with evaluation (Eval and Apply.)

Chapter 10 describes the I/O facilities. Most LISP programs do not require sophisticated I/O, so this may be skimmed on a first reading. The section dealing with input deals extensively with customizing the scanner and reader, which is only of interest to the sophisticated user.

Chapter 11 describes some miscellaneous useful facilities.

Chapter 12 contains the bibliography.

Chapter 13 is an alphabetical index of all functions defined in the manual. Chapter 14 contains an alphabetical index of all global variables and switches defined in the manual.

3 DATA TYPES

3.1 Data Types and Structures Supported in PSL

3.1.1 Data Types

Data objects in PSL are tagged with their type. This means that the type declarations required in many programming languages are not needed. Some functions are "generic" in that the result they return depends on the types of the arguments. A tagged PSL object is called an item, and has a tag field (8 bits on the 68000), an info field (24 bits on the 68000). The info field is either immediate data or an index or address into some other structure (such as the heap or id space). For the purposes of input and output of items, an appropriate notation is used. The full details on syntax is not given in this limited manual - just a subset of the I/O syntax.

The basic data types supported in PSL and a brief indication of their representations are described below.

- **inum** A signed number fitting into info. Inums do not require dynamic storage and are represented in the same form as machine integers. (24 bits [-2**23 .. 2**23 - 1].)

- **id** An identifier (or id) is an item whose info field points to a four-item structure containing the print name, property cell, value cell, and function cell. This structure is contained in the id space. The notation for an id is its print name, an alphanumeric character sequence starting with a letter. One always refers to a particular id by giving its print name. When presented with an appropriate print name, the PSL reader will find a unique id to associate with it. See Chapter 4 for more information on ids. NIL and T are treated as special ids in PSL.

- **pair** A primitive two-item structure which has a left and right part. A notation called dot-notation is used, with the form: (<left-part > . <right-part >). The <left-part > is known as the Car portion and the <right-part > as the Cdr portion. The parts may be any item.

- **vector** A primitive uniform structure of items; an integer index is used to access random values in the structure. The individual elements of a vector may be any item. Access to vectors is by means of functions for indexing, extraction and concatenation, defined in Chapter 6. In the notation for vectors, the elements of a vector are surrounded by square brackets: [item-0 item-1 ... item-n].

- **string** A packed vector (or byte vector) of characters; the elements are small integers representing the ASCII codes for the characters (usually inums). The elements may be accessed by indexing and concatenation functions, defined in Chapter 6. String notation consists of a series of characters enclosed in double quotes, as in "THIS IS A STRING". A quote is included by doubling

it, as in "HE SAID, ""LISP""". (Input strings may cross the end-of-line boundary.)

- **code-pointer** This item is used to refer to the entry point of compiled functions (exprs, fexprs, macros, etc.), permitting compiled functions to be renamed, passed around anonymously, etc. New code-pointers are created by the compiler, which is not available with Mac PSL. They can be printed; the printing function prints the type tag (15), follwed by the address of the entry point. The value appears as (# <15:nnnn > nnnn is the entry point).

3.1.2 Other Notational Conventions

Certain functional arguments can be any of a number of types. For convenience, we give these commonly used sets a name. We refer to these sets as "classes" of primitive data types. In addition to the types described above and the names for classes of types given below, we use the following conventions in the manual. {XXX, YYY} indicates that either data type XXX or data type YYY will do. {XXX}-{YYY} indicates that any object of type XXX can be used except those of type YYY; in this case, YYY is a subset of XXX. For example, {integer, id} indicates that either an integer or an id is acceptable; {any}-{vector} means any type except a vector.

- **any** Any of the types given above. S-expression is another term for any. All PSL entities have some value unless an error occurs during evaluation.
- **atom** The class {any}-{pair}.
- **boolean** The class of global variables {T, NIL}, or their respective values, {T, NIL}. (See Chapter 4.6).
- **character** Integers in the range of 0 to 127 representing ASCII character codes. These are distinct from single-character ids.
- **constant** The class of {integer, string, vector, code-pointer}. A constant evaluates to itself (see the definition of Eval in Chapter 9).
- **extra-boolean** Any value in the system. Anything that is not NIL has the boolean interpretation T.
- **ftype** The class of definable function types. The set of ids {expr, fexpr, macro, nexpr}. The ftype is ONLY an attribute of identifiers, and is not associated with either executable code (code-pointers) or lambda expressions.
- **number** The class of integer.
- **x-vector** Any kind of vector; i.e., a string or vector. word.
- **Undefined** An implementation-dependent value returned by some low-level functions; i.e., the user should not depend on this value.

- **None Returned** A notational convenience used to indicate control functions that do not return directly to the calling point, and hence do not return a value. (e.g., Go)

3.1.3 Structures

Structures are entities created using pairs. Lists are structures very commonly required as parameters to functions. If a list of homogeneous entities is required by a function, this class is denoted by xxx-list, in which xxx is the name of a class of primitives or structures. Thus a list of ids is an id-list, a list of integers is an integer-list, and so on.

- **list** A list is recursively defined as NIL or the pair (any . list). A special notation called list-notation is used to represent lists. List-notation eliminates the extra parentheses and dots required by dot-notation, as illustrated below. List-notation and dot-notation may be mixed, as shown in the second and third examples.

```
dot-notation                    list-notation
(a . (b . (c . NIL)))           (a b c)
(a . (b . c))                   (a b . c)
(a . ((b . c) . (d . NIL)))     (a (b . c) d)
```

```
Note: () is an alternate input representation of NIL.
```

- **a-list** An a-list, or association list, is a list in which each element is a pair, the Car part being a key associated with the value in the Cdr part.

- **form** A form is an S-expression (any) which is legally acceptable to Eval; that is, it is syntactically and semantically accepted by the interpreter or the compiler. (See Chapter 9 for more details.)

- **lambda** A lambda expression must have the form (in list-notation): (lambda parameters . body). "Parameters" is an id-list of formal parameters for "body", which is a form to be evaluated (note the implicit ProgN). The semantics of the evaluation are defined by the Eval function (see Chapter 9).

- **function** A lambda, or a code-pointer. A function is always evaluated as Eval, Spread.

3.2 Predicates Useful with Data Types

Most functions in this section return T if the condition defined is met and NIL if it is not. Exceptions are noted. Defined are type-checking functions and elementary comparisons.

3.2.1 Functions for Testing Equality

Functions for testing equality are listed below. For other functions comparing arithmetic values see Chapter 3.

(Eq U:any V:any): boolean	*expr*

Returns T if U points to the same object as V, i.e., if they are identical items. Eq is not a reliable comparison between numeric arguments. This function should only be used in special circumstances. Normally, equality should be tested with Equal, described below.

(Equal U:any V:any): boolean	*expr*

Returns T if U and V are the same. Pairs are compared recursively to the bottom levels of their trees. Vectors must have identical dimensions and Equal values in all positions. Strings must have identical characters, i.e. all characters must be of the same case. Code-pointers must have Eq values. Other atoms must be EqN equal. A usually valid heuristic is that if two objects look the same if printed with the function Print, they are Equal. If one argument is known to be an atom, Equal is open-compiled as Eq. For example, if

```
(Setq X '(A B C)) and (Setq Y X) have been executed, then
(EQ X Y) is T
(EQ X '(A B C)) is NIL
(EQUAL X '(A B C)) is T
(EQ 1 1) is T
```

(EqStr U:any V:any): boolean	*expr*

Compare two strings, for exact (Case sensitive) equality. EqStr returns T if U and V are Eq or if U and V are equal strings.

3.2.2 Predicates for Testing the Type of an Object

(Atom U:any): boolean	*expr*

Returns T if U is not a pair.

(ConstantP U:any): boolean	*expr*

Returns T if U is a constant (that is, neither a pair nor an id). Note that vectors are considered constants.

(Null U:any): boolean	*expr*

Returns T if U is NIL. This is exactly the same function as Not, defined in Section 2.2.3. Both are available solely to increase readability.

(NumberP U:any): boolean	*expr*

Returns T if U is a number (integer).

(PairP U:any): boolean	*expr*

Returns T if U is a pair.

3.2.3 Boolean Functions

Boolean functions return NIL for "false"; anything non-NIL is taken to be true, although a conventional way of representing truth is as T. Note that T always evaluates to itself. NIL may also be represented as '(). The Boolean functions And, Or, and Not can be applied to any LISP type, and are not bitwise functions. And and Or are frequently used in LISP as control structures as well as Boolean connectives (see Section 7.1). For example, the following two constructs will give the same result:

```
(COND ((AND A B C) D))

(AND A B C D)
```

Since there is no specific Boolean type in LISP and since every LISP expression has a value which may be used freely in conditionals, there is no hard and fast distinction between an arbitrary function and a Boolean function. However, the three functions presented here are by far the most useful in constructing more complex tests from simple predicates.

(Not U:any): boolean	*expr*

Returns T if U is NIL. This is exactly the same function as Null, defined in Section 2.2.2. Both are available solely to increase readability.

(And [U:form]): extra-boolean	*fexpr*

And evaluates each U until a value of NIL is found or the end of the list is encountered. If a non-NIL value is the last value, it is returned; otherwise NIL is returned. Note that And called with zero arguments returns T.

(Or [U:form]): extra-boolean	*fexpr*

U is any number of expressions which are evaluated in order of their appearance. If one is found to be non-NIL, it is returned as the value of Or. If all are NIL, NIL is returned. Note that if Or is called with zero arguments, it returns NIL.

3.3 Converting Data Types

The following functions are used in converting data items from one type to another. They are grouped according to the type returned.

(Intern U:{id,string}): id	*expr*

Gets an id on the id-hash-table. The argument may be an id. Intern searches the id-hash-table for an id with the same print name as U and returns the id on the id-hash-table if a match is found. (See Chapter 4 for a discussion of the id-hash-table. Any properties and GLOBAL values associated with the uninterned U are lost. If U does not match any entry, a new one is created and returned. The argument may also be a string in which case

an identifier in the id-hash-table is looked up, created if necessary, and returned. Note carefully: The id returned from Interning a string has exactly the same print name as the string. Most identifiers have uppercase print names (even if you type in lower case!), but interning "abc" yields an id with a lower case print name.

```
(EQ (INTERN "abc") 'abc) = NIL
```

The maximum number of characters in any token is 80.

(Id2String D:id): string	*expr*

Get name from id space. Id2String returns the Print name of its argument as a string. This is not a copy, so destructive operations should not be performed on the result.

```
(Id2String 'String)  returns "STRING"
```

(List2Vector L:list): vector	*expr*

Copy the elements of the list into a vector of the same Size.

```
(List2Vector '(V E C T O R)) returns [V E C T O R]
```

4 NUMBERS AND ARITHMETIC FUNCTIONS

Most of the arithmetic functions in PSL expect numbers as arguments. In all cases an error occurs if the parameter to an arithmetic function is not a number:

```
***** Non-numeric argument in arithmetic
```

Exceptions to the rule are noted.

The underlying machine arithmetic requires parameters to be either all integers.

4.1 Arithmetic Functions

The functions described below handle arithmetic operations. Please note the remarks at the beginning of this Chapter regarding the mixing of argument types.

(Add1 U:number): number	*expr*

Returns the value of U plus 1; the returned value is of the same type as U.

(- [U:number]): number	*expr*

Returns the difference of all its arguments (the first argument minus all of the remaining arguments). "-" may be called with only one argument. In this case it returns the value of its argument. If "-" is called with no arguments, it returns -1.

(difference U:number V:number): number	*expr*

Two operand version of -.

(Minus U:number): number	*expr*

Returns -U.

(+ [U:number]): number	*macro*

Forms the sum of all its arguments. "+" may be called with only one argument. In this case it returns its argument. If "+" is called with no arguments, it returns zero.

(Plus2 U:number V:number): number	*expr*

Returns the sum of U and V.

(/ [U:number]): number	*expr*

The Quotient of each element in left to right order. Division of two positive or two negative integers is conventional. If exactly one of U and V is negative, the value returned is truncated toward 0. "/" with zero arguments returns 1, with one value returns that value. An error occurs if division by zero is attempted:

```
*** ERROR *** - Divide by Zero
```

(Remainder U:integer V:integer): integer	*expr*

The result is the integer remainder of U divided by V. The sign of the result is the same as the sign of the dividend (U).

```
*** ERROR *** Divide by Zero.
```

(Sub1 U:number): number	*expr*

Returns the value of U minus 1.

([U:number]): number*	*macro*

Returns the product of all its arguments. "*" may be called with only one argument. In this case it returns the value of its argument. If "*" is called with no arguments, it returns 1.

(Times2 U:number V:number): number	*expr*

Returns the product of U and V.

4.2 Functions for Numeric Comparison

The following functions compare the values of their arguments. For functions testing equality (or non-equality) see Section 2.2.1.

(>= U:any V:any): boolean	*expr*

Returns T if U >= V, otherwise returns NIL.

(geq U:any V:any): boolean	*expr*

Same as >=.

(> U:number V:number): boolean	*expr*

Returns T if U is strictly greater than V, otherwise returns NIL.

(greaterp U:number V:number): boolean	*expr*

Same as >.

(<= U:number V:number): boolean	*expr*

Returns T if U <= V, otherwise returns NIL.

(leq U:number V:number): boolean	*expr*

Same as <=.

(< U:number V:number): boolean	*expr*

Returns T if U is strictly less than V, otherwise returns NIL.

(lessp U:number V:number): boolean	*expr*

Same as <.

(MinusP U:any): boolean	*expr*

Returns T if U is a number and less than 0. If U is not a number or is a positive number, NIL is returned.

5 IDENTIFIERS

5.1 Introduction

In PSL variables are called identifiers or ids. An identifier is implemented as a tagged data object (described in Chapter 2) containing a pointer or offset into a four item structure - the id space. One item in this structure is called the print name, which is the external representation of the id.

The interpreter uses an id hash table to get from the print name of an identifier to its entry in the id space. The id space and the id hash table are described below.

5.2 Fields of Ids

An id is an item with an info field; the info field is an offset into a special id space consisting of structures of four fields. The fields (items) are:

- **print-name** The print name points at a string of characters which is the external representation of the identifier. The syntax for identifiers is described in Section 10.2 on reading functions.
- **value-cell** The value of the identifier or a pointer to the value in the heap is stored in this field. If no value exists, this cell contains an unbound identifier indicator. These cells can be accessed by functions defined in this chapter.
- **property-cell** A list of properties (indicators and values), accessed with Put and Get.
- **function-cell** An id may have a function or macro associated with it. Access is by means of the PutD, GetD, and RemD functions.

5.3 Identifiers and the Id hash table

The method used by PSL to retrieve information about an identifier makes use of the id hash table (corresponding to the Oblist, or Object list, in some versions of LISP). A hash function is applied to the identifier name giving a position in the id hash table. The contents of the hash table at that point contain an offset into the id space. For a new identifier, the next free position in the id space is found and a pointer to it is placed in the hash table entry.

The process of putting an id into the hash table is called interning. This is done automatically by the LISP reader, so any id typed in at the terminal is interned. Interning can also be done by the programmer using the function Intern to convert a string to an id. An id may have an entry in the id space without being interned. In fact it is possible to have several ids with the same print name, one interned and the others not. However, in the current Mac PSL, non interned id's are not present, and the id hash table is not actually used.

Note that when one starts PSL, the id space already contains approximately 265 ids. These include all of the ASCII characters, the functions and globals described in this manual, plus system functions and globals. If a user uses any of these names for his own functions or globals, there can be a conflict. A warning message appears if a user tries to redefine a system function.

```
Warning - Function Foo Redefined
```

In version 1.0 of Mac PSL, there are 100 id's available for definition by the user.

Information on converting ids to other types can be found in Chapter 10 and Section 2.3.

5.3.1 Identifier Functions

The following function deals with identifiers and the id hash table.

(MapObl FNAME:function): Undefined	*expr*

MapObl applies function FNAME to each id interned in the current hash table.

5.4 Property List Functions

The property cell of an identifier points to a "property list". The list is used to quickly associate an id name with a set of entities; those entities are called "flags" if their use gives the id a boolean value, and "properties" if the id is to have an arbitrary attribute (an indicator with a property).

(Put U:id IND:id PROP:any): any	*expr*

The indicator IND with the property PROP is placed on the property list of the id U. If the action of Put occurs, the value of PROP is returned.

```
(Put 'Jim 'Height 68)
```

The above returns 68 and places (Height . 68) on the property list of the id Jim.

(Get U:id IND:id): any	*expr*

Returns the property associated with indicator IND from the property list of U. If U does not have indicator IND, NIL is returned. (In older LISPs, Get could access functions.) Get returns NIL if U is not an id.

```
(Get 'Jim 'Height) returns 68
```

(RemProp U:id IND:id): any	*expr*

Removes the property with indicator IND from the property list of U. Returns the removed property or NIL if there was no such indicator.

5.4.1 Direct Access to the Property Cell

Allows access to the entire property list for a particular id.

(Prop U:id): any	*expr*

Returns the property list of U.

5.5 Value Cell Functions

The contents of the value cell may be accessed by Eval (Chapter 9) and changed by SetQ or sometimes Set.

(SetQ VARIABLE:id VALUE:any): any	*fexpr*

The value of the current binding of VARIABLE is replaced by the value of VALUE.

```
(SETQ X 1)

is equivalent to

(SET 'X 1)
```

(Set EXP:id VALUE:any): any	*expr*

EXP must be an identifier or a type mismatch error occurs. The effect of Set is replacement of the item bound to the identifier by VALUE. If the identifier is not a LOCAL variable or has not been declared GLOBAL, it is automatically declared FLUID.

```
EXP must not evaluate to T or NIL or an error occurs:

***** Cannot change T or NIL
```

(SetF [LHS:form RHS:any]): RHS:any	*macro*

Currently the same as SetQ. Later enhancement will provide full SetF capabilities. *Warning - It is exactly the same as SetQ, there is no special processing*

5.6 System Global Variables, Switches and Other "Hooks"

5.6.1 Introduction

A number of global variables provide global control of the LISP system, or implement values which are constant throughout execution. Certain options are controlled by switches, with T or NIL properties (e.g., Printing the results of an evaluated form with *PVAL); others require a value, such as the count of garbage collections. PSL has the convention (following the REDUCE/RLISP convention) of using a "!*" in the name of the variable: !*xxxxx for GLOBAL variables expecting a T/NIL value (called "switches"), and xxxxx!* for other GLOBALs. Chapter 14 is an index of switches and global variables used in PSL.

5.6.2 Special Global Variables

NIL [Initially: NIL]	*global*

NIL is a special GLOBAL variable. It is protected from being modified by Set or SetQ.

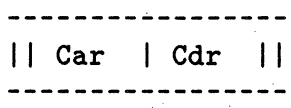
T [Initially: T]	*global*

T is a special GLOBAL variable. It is protected from being modified by Set or SetQ.

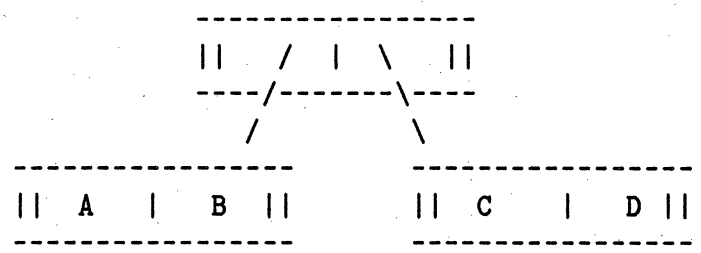
6 LIST STRUCTURE

6.1 Introduction to Lists and Pairs

The pair is a fundamental PSL data type, and is one of the major attractions of LISP programming. A pair consists of a two-item structure. In PSL the first element is called the Car and the second the Cdr; in other LISPs, the physical relationship of the parts may be different. An illustration of the tree structure is given below as a box diagram; the Car and the Cdr are each represented as a portion of the box.



As an example, a tree written as ((A . B) . (C . D)) in dot-notation is drawn below as a box diagram.



The box diagrams are tedious to draw, so dot-notation is normally used. Note that a space is left on each side of the . to ensure that pairs are not confused with floats. Note also that in RLISP a dot may be used as the infix operator for the function Cons, as in the expression x := 'y . 'z;, or as part of the notation for pairs, as in the expression x := '(y . z).

An important special case occurs frequently enough that it has a special notation. This is a list of items, terminated by convention with the id NIL. The dot and surrounding parentheses are omitted, as well as the trailing NIL. Thus

```
(A . (B . (C . NIL)))
```

can be represented in list-notation as

```
(A B C)
```

6.2 Basic Functions on Pairs

The following are elementary functions on pairs. All functions in this chapter which require pairs as parameters signal a type mismatch error if the parameter given is not a pair.

(Cons U:any V:any): pair	*expr*

Returns a pair which is not Eq to anything else and has U as its Car part and V as its Cdr part. In RLISP syntax the dot, ".", is an infix operator meaning Cons. Thus (A . (B . fn C) . D) is equivalent to Cons (A, Cons (Cons (B, fn C), D)).

(Car U:pair): any	*expr*

The left part of U is returned. A type mismatch error occurs if U is not a pair, except when U is NIL. Then NIL is returned. (Car (Cons a b)) == > a.

(Cdr U:pair): any	*expr*

The right part of U is returned. A type mismatch error occurs if U is not a pair, except when U is NIL. Then NIL is returned. (Cdr (Cons a b)) == > b.

Only the composites of Car and Cdr below are supported:

```
                Car                                   Cdr
      Caar              Cdar                Cadr              Cddr
                      Caddr  Cdddr
                     Cadddr  Cddddr
```

These are all exprs of one argument. They may return any type. An example of their use is that Caddr p is equivalent to Car Cdr Cdr p. As with Car and Cdr, a type mismatch error occurs if the argument does not possess the specified component.

(NCons U:any): pair	*expr*

Cons a Nil onto the end of U.

(XCons U:any V:any): pair	*expr*

Equivalent to Cons (V, U).

(RplacA U:pair V:any): pair	*expr*

The Car of the pair U is replaced by V, and the modified U is returned. (If U is (a . b) then (V .b) is returned). A type mismatch error occurs if U is not a pair.

(RplacD U:pair V:any): pair	*expr*

The Cdr of the pair U is replaced by V, and the modified U is returned. (If U is (a . b) then (a . V) is returned). A type mismatch error occurs if U is not a pair.

6.3 Functions for Manipulating Lists

The following functions are meant for the special pairs which are lists, as described in Section 5.1. Note that the functions described in Chapter 6 can also be used on lists.

6.3.1 Membership and Length of Lists

(MemQ A:any B:list): extra-boolean	*expr*

Is A a member of the list B, using an Eq check is used for comparison.

```
(Defun Memq (A L)
   (Cond((Null L) Nil)
```

```
        ((Eq A (First L)) L)
        (T (Memq A (Rest L)))))
```

(Length X:any): integer	*expr*

The top level length of the list X is returned.

```
(Defun Length (X)
   (Cond ((Atom X) 0)
         (T (Plus (Length (Rest X)) 1))))
```

6.3.2 Constructing, Appending, and Concatenating Lists

(List [U:any]): list	*fexpr*

Construct a list of the evaluated arguments. A list of the evaluation of each element of U is returned.

(Append U:list V:list): list	*expr*

Returns a constructed list in which the last element of U is followed by the first element of V. The list U is copied, but V is not.

```
(Defun Append (U V)
   (Cond ((Null U) V)
         (T (Cons (Car U) (Append (Cdr U) V)))))
```

6.3.3 Deleting Elements of Lists

(DelatQ U:any V:a-list): a-list	*expr*

Delete first (U . xxx) from V, using Eq to check equality with U.

6.3.4 List Reversal

(Reverse U:list): list	*expr*

Returns a copy of the top level of U in reverse order.

```
(Defun Reverse (U)
  (let (W)
    (While U
      (ProgN
        (Setq W (Cons (Car U) W))
        (Setq U (Cdr U))))
    W))
```

(ReversIP U:list): list	*expr*

Destructive Reverse.

6.4 Functions for Building and Searching A-Lists

(Atsoc R1:any R2:any): any	*expr*

Scan R2 for pair with Car Eq R1.

(Pair U:list V:list): a-list	*expr*

U and V are lists which must have an identical number of elements. If not, an error occurs. Returned is a list in which each element is a pair, the Car of the pair being from U and the Cdr being the corresponding element from V.

```
(De Pair (U V)
       (Cond ((And U V)(Cons (Cons (Car U)(Car V))
                             (Pair (Cdr U)(Cdr V))))
             ((Or U V)(ERROR 000 "Different length lists"))
             (T Nil)))
```

7 STRINGS AND VECTORS

7.1 Vector-Like Objects

In this chapter, LISP strings, vectors, word-vectors, halfword-vectors, and byte-vectors are described. Each may have several elements, accessed by an integer index. For convenience, members of this set are referred to as x-vectors. X-vector functions also apply to lists. Currently, the index for x-vectors ranges from 0 to an upper limit, called the Size or UpB (upper bound).

7.2 Vectors

A vector is a structured entity in which random item elements may be accessed with an integer index. A vector has a single dimension. Its maximum size is determined by the implementation and available space. A suggested input/output "vector notation" is defined (see Chapter 10).

(iGetV V:vector INDEX:integer): any	*expr*

Returns the value stored at position INDEX of the vector V. If V not a vector, the system will error (it does no type checking).

(MkVect UPLIM:integer): vector	*expr*

Defines and allocates space for a vector with UPLIM + 1 elements accessed as 0...UPLIM. Each element is initialized to NIL. If UPLIM is -1, an empty vector is returned. An error occurs if UPLIM is < -1 or if there is not enough space for a vector of this size:

```
***** A vector of size UPLIM cannot be allocated
```

(iPutV V:vector INDEX:integer VALUE:any): any	*expr*

Stores VALUE in the vector V at position INDEX. VALUE is returned. It does not check to see if V is a vector!!! Be careful.

(ISizeV U:any): {NIL,integer}	*expr*

Returns the upper limit of Vector U.

7.3 General X-Vector Operations

(Concat X:x-vector Y:x-vector): x-vector	*expr*

Concatenate two x-vectors. Currently they must be of same type. Will only work for strings in MacPSL.

8 FLOW OF CONTROL

8.1 Conditionals

8.1.1 Conds

(Cond [U:form-list]): any	*fexpr*

The LISP function Cond corresponds to the If statement of most programming languages.

The arguments to Cond have the form:

```
(COND (predicate action action ...)
      (predicate action action ...)
      ...
      (predicate action action ...) )
```

The predicates are evaluated in the order of their appearance until a non-NIL value is encountered. The corresponding actions are evaluated and the value of the last becomes the value of the Cond. If there are no corresponding actions, the value of the predicate is returned.

8.2 Sequencing Evaluation

These functions provide for explicit control sequencing, and the definition of blocks altering the scope of local variables.

(ProgN [U:form]): any	*fexpr*

U is a set of expressions which are executed sequentially. The value returned is the value of the last expression.

8.3 Iteration

(While E:form [S:form]): NIL	*macro*

This is the most commonly used construct for indefinite iteration in LISP. E is evaluated; if non-NIL, the S's are evaluated from left to right and then the process is repeated. If E evaluates to NIL the While returns NIL.

(MapC X:list FN:function): NIL	*expr*

FN is applied to successive Car segments of list X. NIL is returned.

(Let A:list [B:form]): any	*macro*

Let is a macro giving a more perspicuous form for writing lambda expressions. The basic form is

```
(LET ((V1 I1) (V2 I2) ...(Vn In)) S1 S2 ...  Sn)
```

The I's are evaluated (in an unspecified order), and then the V's are bound to these values, the S's evaluated, and the value of the last is returned. Note that the I's are evaluated in the outer environment before the V's are bound.

8.4 Non-Local Exits

The non-local exit constructs Catch and Throw allow exit from forms without following the standard return convention defined within the form (i.e. terminate a form early). This should not, however, be used indiscriminately. The lexical restrictions on their more local counterparts ensure that the flow of control can be ascertained by looking at a single piece of code. With Catch and Throw, control may be passed to and from totally unrelated pieces of code. Under some conditions, these functions are invaluable. Under others, they can wreak havoc.

(Catch TAG:id [FORM:form]): any	*fexpr*

Catch evaluates the TAG and then calls Eval on the FORMs in a protected environment. If during this evaluation (Throw TAG VAL) occurs, Catch immediately returns VAL. If no Throw occurs, the value of the last FORM is returned. Note that in general only Throws with the same TAG are caught. Throws whose TAG is not Eq to that of Catch are passed on out to surrounding Catches. A TAG of NIL, however, is special. (Catch NIL FORM) catches any Throw. The tag of NIL serves to match any tag specified to Throw.

(Throw TAG:id VAL:any): None Returned	*expr*

This passes control to the closest surrounding Catch with an Eq or null TAG. If there is no such surrounding Catch it is an error in the context of the Throw. That is, control is not Thrown to the top level before the call on Error. (Non-local Goto.)

Some examples:

With

```
(DE DOIT (x)
 (COND ((EQN x 1) 100)
       (T (THROW 'FOO 200))))

(CATCH 'FOO (DOIT 1) (PRINT  "NOPE") 0)
```

will continue and execute the PRINT statement and return 0

while

```
(CATCH 'FOO (DOIT 2) (PRINT  "NOPE") 0)
```

will of course THROW, returning 200 and not executing the last forms.

9 FUNCTION DEFINITION AND BINDING

9.1 Function Definition in PSL

Functions in PSL are GLOBAL entities. To avoid function-variable naming clashes, the Standard LISP Report required that no variable have the same name as a function. There is no conflict in PSL, as separate function cells and value cells are used. A warning message is given for compatibility. The first major section in this chapter describes how to define new functions; the second describes the binding of variables in PSL.

9.2 Function Types

Eval-type functions are those called with evaluated arguments. NoEval functions are called with unevaluated arguments. Spread-type functions have their arguments passed in a one-to-one correspondence with their formal parameters. NoSpread functions receive their arguments as a single list.

There are four function types implemented in PSL:

- **expr** An Eval, Spread function, with a maximum of 15 arguments. In referring to the formal parameters we mean their values. Each function of this type should always be called with the expected number of parameters, as indicated in the function definition. Future versions of PSL will check this consistency.

- **fexpr** A NoEval, NoSpread function. There is no limit on the number of arguments. In referring to the formal parameters we mean the unevaluated arguments, collected as a single List, and passed as a single formal parameter to the function body.

- **nexpr** An Eval, NoSpread function. Each call on this kind of function may present a different number of arguments, which are evaluated, collected into a list, and passed in to the function body as a single formal parameter.

- **macro** The macro is a function which creates a new S-expression for subsequent evaluation or compilation. There is no limit to the number of arguments a macro may have. The description of the Eval function in Chapter 9 provide precise details.

9.2.1 Notes on Code Pointers

A code-pointer may be displayed by the Print functions. The value appears in the convention of the implementation (# <15:nnnn >, and nnnn is the function's entry point. A code-pointer may not be created by Compress. (See Chapter 10 for descriptions of Explode and Compress.) The code-pointer associated with a compiled function may be retrieved by GetD and is valid as long as PSL is in execution (on the 68000, compiled code is not relocated, so code-pointers do not change). A code-pointer may be stored using PutD, Put, SetQ and the like or by being bound to a variable. It may be checked for equivalence by Eq.

9.2.2 Functions Useful in Function Definition

In PSL, ids have a function cell that usually contains an executable instruction which either JUMPs directly to the entry point of a compiled function or executes a CALL to an auxiliary routine that handles interpreted functions, undefined functions, or other special services. The user can pass anonymous function objects around either as a code-pointer, which is a tagged object referring to a compiled code block, or a lambda expression, representing an interpreted function.

(PutD FNAME:id TYPE:ftype BODY:{lambda,code-pointer}): id	*expr*

Creates a function with name FNAME and type TYPE, with BODY as the function definition. If successful, PutD returns the name of the defined function.

If the body is a code-pointer a special instruction to jump to the start of the code is placed in the function cell. If it is a lambda, the lambda expression is saved on the property list under the indicator !*LAMBDALINK and a call to an interpreter function (LambdaLink) is placed in the function cell.

The TYPE is recorded on the property list of FNAME if it is not an expr.

After using PutD on FNAME, GetD returns a pair of the the FNAME's (TYPE . BODY).

(GetD U:any): {NIL, pair}	*expr*

If U is not the name of a defined function, NIL is returned. If U is a defined function then the pair ({expr, fexpr, macro, nexpr} . {code-pointer, lambda}) is returned.

(CopyD NEW:id OLD:id): NEW:id	*expr*

The function body and type for NEW become the same as OLD. If no definition exists for OLD an error:

```
(OLD "has no definition in COPYD")
```

is given. NEW is returned.

9.2.3 Function Definition in LISP Syntax

The functions Defun, De, Df, Dn, Dm, and Ds are most commonly used in the LISP syntax form of PSL.

(Defun FNAME:id PARAMS:id-list [FN:form]): id	*macro*

Defines the function named FNAME, of type expr. The forms FN are made into a lambda expression with the formal parameter list PARAMS, and this is used as the body of the function.

Previous definitions of the function are lost. The name of the defined function, FNAME, is returned.

(De FNAME:id PARAMS:id-list [FN:form]): id	*macro*

Exactly the same as Defun.

(Df FNAME:id PARAM:id-list FN:any): id	*macro*

Defines the function named FNAME, of type fexpr. The forms FN are made into a lambda expression with the formal parameter list PARAMS, and this is used as the body of the function.

Previous definitions of the function are lost. The name of the defined function, FNAME, is returned.

(Dn FNAME:id PARAM:id-list FN:any): id	*macro*

Defines the function named FNAME, of type nexpr. The forms FN are made into a lambda expression with the formal parameter list PARAMS, and this is used as the body of the function.

Previous definitions of the function are lost. The name of the defined function, FNAME, is returned.

(Dm MNAME:id PARAM:id-list FN:any): id	*macro*

Defines the function named FNAME, of type macro. The forms FN are made into a lambda expression with the formal parameter list PARAMS, and this is used as the body of the function.

Previous definitions of the function are lost. The name of the defined function, FNAME, is returned.

9.2.4 Low Level Function Definition Primitives

The following function is used especially by PutD and GetD, defined above in Section 8.2.2, and by Eval and Apply, defined in Chapter 9.

(Code!-Number!-Of!-Arguments C:code-pointer): {NIL,integer}	*expr*

Some compiled functions have the argument number they expect stored in association with the codepointer C. This integer, or NIL is returned.

9.3 Variables and Bindings

Variables in PSL are ids, and associated values are usually stored in and retrieved from the value cell of this id. If variables appear as parameters in lambda expressions or in let's, the contents of the value cell are saved on a binding stack. A new value or NIL is stored in the value cell and the computation proceeds. On exit from the lambda or let the old value is restored. This is called the "shallow binding" model of LISP. It is chosen to permit compiled code to do binding efficiently. For even more efficiency, compiled code may eliminate the variable names and simply keep values in registers or a stack. The scope of a variable is the range over which the variable has a defined value. There are three different binding mechanisms in PSL.

- **LOCAL BINDING** Only compiled functions bind variables locally. Local variables occur as formal parameters in lambda expressions and as LOCAL

variables in let's. The binding occurs as a lambda expression is evaluated or as a let form is executed. The scope of a local variable is the body of the function in which it is defined.

- **FLUID BINDING** FLUID variables are GLOBAL in scope but may occur as formal parameters or Let form variables. In interpreted functions, all formal parameters and LOCAL variables are considered to have FLUID binding until changed to LOCAL binding by compilation. A variable can be treated as a FLUID only by declaration. If FLUID variables are used as parameters or LOCALs they are rebound in such a way that the previous binding may be restored. All references to FLUID variables are to the currently active binding. Access to the values is by name, going to the value cell.

- **GLOBAL BINDING** GLOBAL variables may never be rebound. Access is to the value bound to the variable. The scope of a GLOBAL variable is universal. Variables declared GLOBAL may not appear as parameters in lambda expressions or as let form variables. A variable must be declared GLOBAL prior to its use as a GLOBAL variable since the default type for undeclared variables is FLUID. Note that the interpreter does not stop one from rebinding a global variable. The compiler will issue a warning in this situation.

9.3.1 Binding Type Declaration

(Fluid IDLIST:id-list): NIL	*expr*

The ids in IDLIST are declared as FLUID type variables (ids not previously declared are initialized to NIL). Variables in IDLIST already declared FLUID are ignored. Changing a variable's type from GLOBAL to FLUID is not permissible and results in the error:

```
***** ID cannot be changed to FLUID
```

(Global IDLIST:id-list): NIL	*expr*

The ids of IDLIST are declared GLOBAL type variables. If an id has not been previously declared, it is initialized to NIL. Variables already declared GLOBAL are ignored. Changing a variable's type from FLUID to GLOBAL is not permissible and results in the error:

```
***** ID cannot be changed to GLOBAL
```

(UnFluid IDLIST:id-list): NIL	*expr*

The variables in IDLIST which have been declared as FLUID variables are no longer considered as FLUID variables. Others are ignored. This affects only compiled functions, as free variables in interpreted functions are automatically considered FLUID (see [Griss 81]).

10 THE INTERPRETER

10.1 Evaluator Functions Eval and Apply

The PSL evaluator uses an identifier's function cell to access the address of the code for executing the identifier's function definition, as described in chapter 8. The function cell contains either the entry address of a compiled function, or the address of a support routine that either signals an undefined function or calls the lambda interpreter. The PSL model of a function call is to place the arguments (after treatment appropriate to function type) in "registers", and then to jump to or call the code in the function cell.

Expressions which can be legally evaluated are called forms. They are restricted S-expressions:

```
form ::=  id
        | constant
        | (id form ... form)
        | (special . any)    % Special cases: COND, etc.
                             % usually fexprs or macros.
```

The definitions of Eval and Apply may clarify which expressions are forms.

In Eval, Apply, and the support functions below, ContinuableError is used to indicate malformed lambda expressions, undefined functions or mismatched argument numbers; the user is permitted to correct the offending expression or to define a missing function inside a Break loop.

The functions Eval and Apply are central to the PSL interpreter. Since their efficiency is important, some of the support functions they use are hand-coded in LAP. The functions LambdaApply, LambdaEvalApply, CodeApply, CodeEvalApply, and IDApply1 are support functions for Eval and Apply. CodeApply and CodeEvalApply are coded in LAP. IDApply1 is handled by the compiler.

(Eval U:form): any	*expr*

The value of the form U is computed. The following is an approximation of the real code, leaving out some implementation details.

```
(DEFUN EVAL (U)
  (let (FN)
    (COND
      ((IDP U) (VALUECELL U))
    % ValueCell  returns the contents of Value Cell if ID
    % is bound, else signals unbound error.  Valuecell is not
    % present in Mac PSL.
      ((NOT (PAIRP U)) U)
    % This is a "constant" which EVAL's to itself
      ((EQ (CAR (CAR U)) 'LAMBDA)
          (LAMBDAEVALAPPLY (CAR U) (CDR U)))
    % LambdaEvalApply applies the lambda- expression Car U
    % list containing the evaluation of each argument in C
    % LambdaEvalApply is not accessible to the Mac PSL user.
      ((CODEP (CAR U))
```

```
      (CODEEVALAPPLY (CAR U) (CDR U)))
% CodeEvalApply applies the function with code-pointer
% to the list containing the evaluation of each argume
% Cdr U.
% CodeEvalApply is not accessible to the Mac PSL user.
  ((NOT (IDP (CAR U)))
      % Signal an error.
      (ERROR 1101
         "Ill-formed expression in EVAL" U))
 (T (SETQ FN (GETD (CAR U)))
    (COND
      ((NULL FN)
        % Another Error
        (ERROR 1001 "Undefined function EVAL" ))
      ((EQ (CAR FN) 'EXPR)
        (COND
          ((CODEP (CDR FN))
          % COdeP is not accessible to the Mac PSL User.
          % CodeEvalApply applies the function with
          % codepointer Cdr FN to the list containing
          % evaluation of each argument in Cdr U.
            (CODEEVALAPPLY (CDR FN) (CDR U)))
          (T
            (LAMBDAEVALAPPLY
              (CDR FN) (CDR U)))))))
      % LambdaEvalApply is not accessible to the Mac PSL User.
      % LambdaEvalApply applies the lambda-expression Cdr FN
      % list containing the evaluation of each argument in C
      ((EQ (CAR FN) 'FEXPR)
        % IDApply1 applies the fexpr Car U to the list of
        % unevaluated arguments.
        % IDApply1 is not accessible to the Mac PSL user.
         (IDAPPLY1 (CDR U) (CAR U))))
      ((EQ (CAR FN) 'MACRO)
      % IDApply1 first expands the macro call U and then
      % evaluates the result.
        (EVAL (IDAPPLY1 U (CAR U)))))
      ((EQ (CAR FN) 'NEXPR)
      % IDApply1 applies the nexpr Car U to the list obt
      % by evaluating the arguments in Cdr U.
       (IDAPPLY1 (EVLIS (CDR U)) (CAR U)))))))
```

(Apply FN:id,function ARGS:form-list): any	*expr*

Apply allows one to make an indirect function call. It returns the value of FN with actual parameters ARGS. The actual parameters in ARGS are already in the form required for binding to the formal parameters of FN. PSL permits the application of nexprs and fexprs; the effect is the same as (Apply (Cdr (GetD FN)) ARGS); i.e. no fix-up is done to quote arguments, etc. as in some LISPs. A call to Apply using List on the second argument [e.g. (Apply F (List X Y))] is compiled so that the list is not actually constructed.

The following is an approximation of the real code, leaving out implementation details.

```
(DEFUN APPLY (FN ARGS)
```

```
(LET (DEFN)
  (COND
    ((CODEP FN)
      % Spread the ARGS into the registers and transfer
      % entry point of the function.
      (CODEAPPLY FN ARGS))
    ((EQ (CAR FN) 'LAMBDA)
      % Bind the actual parameters in ARGS to the formal
      % parameters of the lambda expression If the two l
      % are not of equal length then signal
      % (ERROR (LIST 1204
      %          "Number of parameters do not match"
      %          (CONS FN ARGS)))
      (LAMBDAAPPLY FN ARGS))
    ((NOT (IDP FN))
      (ERROR (LIST 1104
                   "Ill-formed function in APPLY"
                   (CONS FN ARG))))
    ((NULL (SETQ DEFN (GETD FN)))
     (ERROR (LIST 1004
                  "Undefined function in Apply"
                  (CONS FN ARGS))))
    (T
      % Do EXPR's, NEXPR's, FEXPR's and MACRO's alike, a
      % EXPR's
      (APPLY (CDR DEFN) ARGS)))))
```

10.2 Support Functions for Eval and Apply

(EvLis U:any-list): any-list	*expr*

EvLis returns a list of the evaluation of each element of U.

(EvProgN U:form-list): any	*expr*

Evaluates each form in U in turn, returning the value of the last. Used for various implied ProgNs.

10.3 Special Evaluator Functions, Quote, and Function

(Quote U:any): any	*fexpr*

Returns U. Thus the argument is not evaluated by Eval.

11 INPUT AND OUTPUT

11.1 Introduction

One category of input and output in LISP is "symbolic" I/O. This allows a user to print or read possibly complex LISP objects with one or a few calls on standard functions. PSL also has powerful general-purpose I/O.

11.1.1 Organization of this Chapter

We first discuss the syntax used for symbolic input and output. The syntax described applies to PSL programs, interactive typein, format of data in data files, and to output by PSL programs except when special formatting is used.

Functions for printing and reading follow. All (textual) input and output functions are discussed.

11.2 Printed Representation of LISP Objects

Most of this section is devoted to the representation of tokens. In addition to tokens there are composite objects with printed representations: lists and vectors. We briefly discuss their printed formats first.

```
"(" expression expression  . . . ")"
"(" expression expression . . . "." expression)
"[" expression expression . . . "]"
```

Of these the first two are for lists. Where possible, the first notation is preferred and the printing routines use it except when the second form is needed. The second form is required when the CDR of a PAIR is neither NIL nor another pair. The third notation is for vectors. For example:

```
(A . (B . C)) % An S-EXPRESSION
(A B . C)     % Same value, list notation
(A B C)       % An ordinary list
[A B C]       % A vector
```

The following standards for representing tokens are used:

- Ids begin with a letter or any character preceded by an escape character. They may contain letters, digits and escaped characters. Ids may also start with a digit, if the first non-digit following is a plus sign, minus sign, or letter other than "b" or "e". This is to allow identifiers such as "1+" which occur in some LISPs.

If !*Raise is non-NIL, unescaped lower case letters are folded to upper case. The maximum size of an id (or any other token) is currently 80 characters.

Note: Using lower case letters in identifiers may cause portability problems. Lower case letters are automatically converted to upper case if the !*RAISE switch is T. This case conversion is done only for id input, not for single character or string input.

The following examples show identifiers in a form accepted by the LISP scan table. Note that most characters are treated as letters by the LISP scan table, while they are treated as delimiters by the RLISP scan table.

- ThisIsALongIdentifier
- THISISALONGIDENTIFIER
- ThisIsALongIdentifierAndDifferentFromTheOther
- this-is-a-long-identifier-with-dashes
- this_is_a_long_identifier_with_underscores
- an-identifier-with-dashes
- *RAISE
- !2222

- Strings begin with a double quote (") and include all characters up to a closing double quote. A double quote can be included in a string by doubling it. An empty string, consisting of only the enclosing quote marks, is allowed. The characters of a string are not affected by the value of the !*RAISE. Examples:

- "This is a string"
- "This is a ""string"""
- ""

- Integers begin with a digit, optionally preceded by a + or - sign, and consist only of digits. The GLOBAL input radix is 10; there is no way to change this. Examples:

- 100
- +5234

- Code-pointers cannot be read directly, but can be printed and constructed. Currently printed as # <15:address >.

- Anything else is printed as # <Unknown:nnnn >, where nnnn is the value found in the argument register. Such items are not legal LISP entities and may cause garbage collector errors if they are found in the heap. They cannot be read in.

11.3 Functions for Printing

11.3.1 Basic Printing

(Prin1 ITM:any): ITM:any	*expr*

Prin1 is the basic LISP printing function. It operates upon well-formed, non-circular (non-self-referential) structures. The result can be parsed by the function Read.

(Prin2 ITM:any): ITM:any	*expr*

Prin2 is similar to Prin1, except that strings are printed without the surrounding double quotes, and delimiters within ids are not preceded by the escape character.

(Print U:any): U:any	*expr*

Display U using Prin1 and terminate line using Terpri.

11.3.2 Whitespace Printing Functions

(TerPri): NIL	*expr*

Terminate OUTPUT line on screen.

11.3.3 The Fundamental Printing Function

(WriteChar CH:character): character	*expr*

Write one character to Screen. All output is defined in terms of this function. If CH is equal to char EOL (ASCII LF, 8#12) the line counter POSN is set to zero. Otherwise, it is increased by one.

11.3.4 Additional Printing Functions

(Prin2L L:any): L	*expr*

Prin2, except that a list is printed without the top level parens.

(Prin2T X:any): any	*expr*

Output X using Prin2 and terminate line with Terpri.

11.4 Functions for Reading

11.4.1 Reading S-Expressions

(Read): any	*expr*

Reads and returns the next S-expression. Valid input forms are: vector-notation, pair-notation, list-notation, numbers, strings, and identifiers. Identifiers are interned (see the Intern function in Chapter 4).

11.4.2 The Fundamental Reading Function

(ReadChar): character	*expr*

Reads one character from the user.

11.4.3 Input Status and Mode

*!*RAISE [Initially: T]*	*switch*

If !*RAISE is non-NIL, all characters input for ids through PSL input functions are raised to upper case. If !*RAISE is NIL, characters are input as is. A string is unaffected by !*RAISE.

12 MISCELLANEOUS USEFUL FEATURES

12.1 Exiting PSL

On the Mac, one simply selects the Quit option in the menu and that will terminate the PSL.

12.2 Garbage Collection

(Reclaim): Undefined	*expr*

Reclaim is the user level call to the garbage collector. Reclaim is is used within the system to call the garbage collector. Active data in the heap is made contiguous and all tagged pointers into the heap from active local stack frames, the binding stack and the symbol table are relocated. If !*GC is T, prints some statistics. Increments GCKNT!*.

*!*GC [Initially: NIL]*	*switch*

!*GC controls the printing of garbage collector messages. If NIL, no indication of garbage collection occurs. If non-NIL various system dependent messages may be displayed.

GCKNT! [Initially: 0]*	*global*

Records the number of times that Reclaim has been called to this point. GCKNT!* may be reset to another value to record counts incrementally, as desired.

13 BIBLIOGRAPHY

The following books and articles either are directly referred to in the manual text, or will be helpful for supplementary reading.

[Allen 79] Allen, J. R.
The Anatomy of LISP.
McGraw-Hill, New York, New York, 1979.

[Baker 78] Baker, H. G.
Shallow Binding in LISP 1.5.
CACM 21(7):565, July, 1978.

[Benson 81] Benson, E. and Griss, M. L.
SYSLISP: A Portable LISP Based Systems Implementation Language.
Utah Symbolic Computation Group Report UCP-81, University of Utah, Department of Computer Science, February, 1981.

[Bobrow 76] Bobrow, R. J.; Burton, R. R.; Jacobs, J. M.; and Lewis, D.
UCI LISP MANUAL (revised).
Online Manual RS:UCLSP.MAN, University of California, Irvine, 1976.

[Charniak 80] Charniak, E.; Riesbeck, C. K.; and McDermott, D. V.
Artificial Intelligence Programming.
Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1980.

[Fitch 77] Fitch, J. and Norman, A.
Implementing LISP in a High Level Language.
Software: Practice and Experience 7, 1977.

[Foderaro 83] Foderaro, J. K., Sklower, K. L. and Layer, K.
The Franz LISP Manual.
University of California, Berkeley, California.
1983.

[Frick 78] Frick, I. B.
Manual for Standard LISP on the DECSYSTEM 10 and 20.
Utah Symbolic Computation Group Technical Report TR-2, University of Utah, Department of Computer Science, July, 1978.

[Griss 77a] Griss, M. L.
BIL: A Portable Implementation Language for LISP-Like Systems.
Utah Symbolic Computation Group Opnote No. 36, University of Utah, Department of Computer Science, 1977.

[Griss 77b] Griss, M. L. and Swanson, M. R.
MBALM/1700 : A Micro-coded LISP Machine for the Burroughs B1726.
In Proceedings of Micro-10 ACM, pages 15. ACM, 1977.

[Griss 78a] Griss, M. L. and Kessler, R. R.
REDUCE 1700: A Micro-coded Algebra System.
In Proceedings of The 11th Annual Microprogramming Workshop, pages 130-138. IEEE, November, 1978.

[Griss 78b] Griss, M. L.
MBALM/BIL: A Portable LISP Interpreter.
Utah Symbolic Computation Group Opnote No. 38, University of Utah, Department of Computer Science, 1978.

[Griss 79a] Griss, M. L.; Kessler, R. R.; and Maguire, G. Q. Jr.
TLISP - A Portable LISP Implemented in P-code.
In Proceedings of EUROSAM 79, pages 490-502. ACM, June, 1979.

[Griss 79b] Griss, M. L. and Kessler, R. R.
A Microprogrammed Implementation of LISP and REDUCE on the Burroughs B1700/B1800 Computer.
Utah Symbolic Computation Group Report UCP 70, University of Utah, Department of Computer Science, 1979.

[Griss 81] Griss, M. L. and Hearn, A. C.
A Portable LISP Compiler.
Software - Practice and Experience 11:541-605, June, 1981.

[Griss 82] Griss, M. L.; Benson. E.; and Hearn, A. C.
Current Status of a Portable LISP Compiler.
In Proceedings of the SIGPLAN 1982 Symposium on Compiler Construction, pages 276-283. ACM SIGPLAN, June, 1982.

[Harrison 73] Harrison, M. C.
Data structures and Programming.
Scott, Foresman and Company, Glenview, Illinois, 1973.

[Harrison 74] Harrison, M. C.
A Language Oriented Instruction Set for BALM.
In Proceedings of SIGPLAN/SIGMICRO 9, pages 161. ACM, 1974.

[Hearn 66] Hearn, A. C.
Standard LISP.
SIGPLAN Notices Notices 4(9), September, 1966.
Also Published in SIGSAM Bulletin, ACM Vol. 13, 1969, p. 28-49.

[Hearn 73] Hearn, A. C.
REDUCE 2 Users Manual.
Utah Symbolic Computation Group, Report UCP-19, University of Utah, Department of Computer Science, 1973.

[Kessler 79] Kessler, R. R.
PMETA - Pattern Matching META/REDUCE.
Utah Symbolic Computation Group, OpNote 40, University of Utah, Department of Computer Science, January, 1979.

[Lefaivre 78] Lefaivre, R.
RUTGERS/UCI LISP MANUAL.
Online Manual, RS:RUTLSP.MAN, Rutgers University, Computer Science Department, May, 1978.

[MACLISP 76]
MACLISP Reference Manual.
Technical Report, MIT, March, 1976.

[Marti 79] Marti, J. B., et al.
Standard LISP Report.
SIGPLAN Notices 14(10):48-68, October, 1979.

[McCarthy 73] McCarthy, J. C. et al.
LISP 1.5 Programmer's Manual.
M.I.T. Press, 1973.
7th Printing January 1973.

[Moore 76] J. Strother Moore II.
The INTERLISP Virtual Machine Specification.
CSL 76-5, Xerox, Palo Alto Research Center, 3333 Coyote Road,etc, September, 1976.

[Nordstrom 73] Nordstrom, M.
A Parsing Technique.
Utah Computational Physics Group Opnote No. 12, University of Utah, Department of Computer Science, November, 1973.

[Nordstrom 78] Nordstrom, M.; Sandewall, E.; and Breslaw, D.
LISP F3 : A FORTRAN Implementation of InterLISP.
Manual, Datalogilaboratoriet, Sturegatan 2 B, S 752 23, Uppsala, SWEDEN, 1978.
Mentioned by M. Nordstrom in 'Short Announcement of LISP F3', a handout at LISP80.

[Norman 81] Norman, A.C. and Morrison, D. F.
The REDUCE Debugging Package.
Utah Symbolic Computation Group Opnote No. 49, University of Utah, Department of Computer Science, February, 1981.

[Pratt 73] Pratt, V.
Top Down Operator Precedence.
In Proceedings of POPL-1. ACM, 1973.

[Quam 69] Quam, L. H. and Diffie, W.
Stanford LISP 1.6 Manual.

Operating Note 28.7, Stanford Artificial Intelligence Laboratory, 1969.

[Sandewall 78] Sandewall, E.
Programming in an Interactive Environment : The LISP Experience.
Computing Surveys 10(1):35-72, March, 1978.

[Steele 84] Steele, G. L.
Common Lisp, The Language
Digital Press, 1984.

[Teitelman 78] Teitelman, W.; et al.
Interlisp Reference Manual, (3rd Revision).
Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto,Calif. 94304, 1978.

[Teitelman 81] Teitleman, W. and Masinter, L.
The InterLISP Programming Environment.
IEEE Computer 14(4):25-34, 1981.

[Terashima 78] Terashima, M. and Goto, E.
Genetic Order and Compactifying Garbage Collectors.
Information Processing Letters 7(1):27-32, 1978.

[Weinreb 81] Weinreb, D. and Moon, D.
LISP Machine Manual.
MIT AI Lab, 1981.
Fourth edition.

[Weissman 67] Weissman.
LISP 1.5 Primer.
Dickenson Publishing Company, Inc., 1967.

[Winston 84] Winston, P. H., and Horn, B. K. P.
LISP, Second Edition.
Addison-Wesley Publishing Company, Reading, Mass., 1984.

14 INDEX OF FUNCTIONS

The following is an alphabetical list of the PSL functions, with the page on which they are defined.

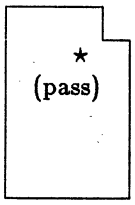
15 INDEX OF GLOBALS AND SWITCHES

The following is an alphabetical list of the PSL global variables, with the page on which they are defined.

THE PORTABLE STANDARD LISP USERS MANUAL

PART 2: UTILITIES

MACINTOSH^{tm} VERSION 1.0



Utah Portable Artificial Intelligence Support Systems Project
Computer Science Department
University of Utah
Salt Lake City, Utah 84112
Version 3.2: 12 April 1984
Modified for Macintosh^{tm} PSL - April 25, 1985

ABSTRACT

This is a description of utilities available to the user of the Macintosh^{tm} version of Portable Standard Lisp (PSL). Since the compiler is not available in this version, the only utilities are defined interpretively.

1 Utilities for Macintosh^{tm} PSL

The *Trace* file defines a simple trace package for Mac PSL. It defines two function TR and UNTR that allows one to trace and untrace a function. Tracing a function, modifies the function definition to print out an informative message upon entry and exit from a function. This information is useful when debugging code and you wish to follow the procedure call tree.

(tr Function-name:id): id	*macro*

Tr will redefine the function so it prints out informative messages upon exit and entry of the function.

(untr Function-name:id): id	*macro*

Untr will untrace a function and restore it to its previous form.

The trace package uses approximately 700 heap items, and thus leaves very little room on the 128K Mac. Therefore, use it with caution on the small version.

2 PSL Lesson Files

Included on the disk is a folder containing 8 PSL lessons. The lessons cover most of the features available to the Mac PSL user. Due to the limitation on the size of files, each lesson had to be broken up into multiple files. The user is encouraged to load in each part of a lesson, evaluate each form in the file and watch the results. The lessons are designed to be self contained, however some parts of a lesson may depend on already evaluating the preceeding parts (i.e. you should evaluate Lessons 3a and 3b before evaluating 3c). We appologize for the terseness of the comments, but once again it is due to memory limitations on the Mac. These lessons, used in conjunction with a good introductory text on Lisp programming, should be sufficient to learn about all of the features of Mac PSL.

3 PSL Demo Files

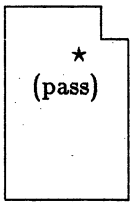
These files are sample Mac PSL programs (all are modifications of programs defined in the text "Lisp" by Winston and Horn) Each program is defined below:

- **Hanoi** - A program to solve the Tower's of Hanoi program
- **Infix** - Translates an infix expression into Lisp prefix notation
- **Matcher** - A simple pattern matcher
- **Translate** - Will translate simple english statements of algebra facts and queries into Lisp forms. This program requires the **Matcher** file to have been loaded first and **will not run on a 128K Mac, due to the lack of available heap space.**

THE PORTABLE STANDARD LISP USERS MANUAL

PART 3: USER GUIDE

MACINTOSH^{tm} VERSION 1.0



Utah Portable Artificial Intelligence Support Systems Project
Computer Science Department
University of Utah
Salt Lake City, Utah 84112
Version 3.2: 12 April 1984
Modified for Macintosh^{tm} PSL - April 25, 1985

ABSTRACT

This is the user's guide for the Macintosh^{tm} version of Portable Standard Lisp (PSL). We describe the features that are Macintosh^{tm} dependent.

Table of Contents

1 Release notes for Macintosh^{tm} Micro-PSL

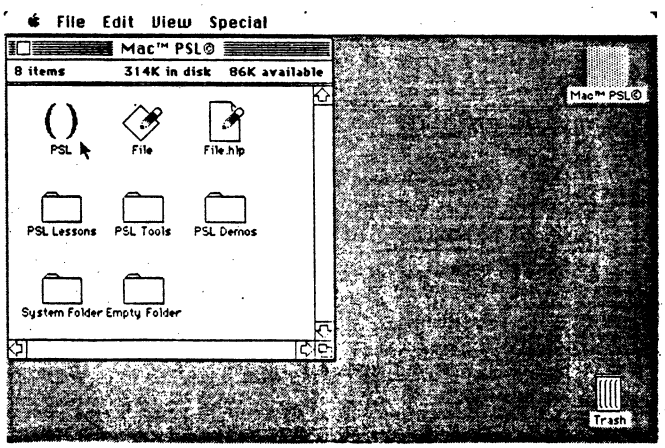
Mac Micro Portable Standard Lisp (PSL) is a small implementation of the Utah Portable Standard Lisp for the Macintosh^{tm}. It has been designed to run on all sizes of Macintosh^{tm} machines. Mac PSL will automatically allocate a heap size based on the amount of available memory. Currently, a 128K machine has about 1200 items available (600 pairs), a 512K Mac with RamDisk has 5000 items, and a 512K Mac without RamDisk, has about 45,000 items. The symbol table is fixed in size and has space available for 100 new symbols. On the 128K Mac, the limited memory space makes the system primarily useful for small educational programs. Part 1 of this manual lists the functions that are available in this version. Note that this is a very small subset of PSL as defined on other implementations.

2 Getting Started

To begin, insert the Mac PSL disk, and open the disk icon. The following files are available on the disk:

- PSL - The Mac PSL program
- File and File.hlp - A public domain text editor and help file. This editor is useful for printing and editing Mac PSL documents
- PSL Lessons Folder - A set of files that may be loaded into Mac PSL that illustrate various features of Mac PSL
- PSL Tools Folder - Files that may be useful in developing PSL programs
- PSL Demos Folder - A set of demo files.
- Standard System and Empty Folder.

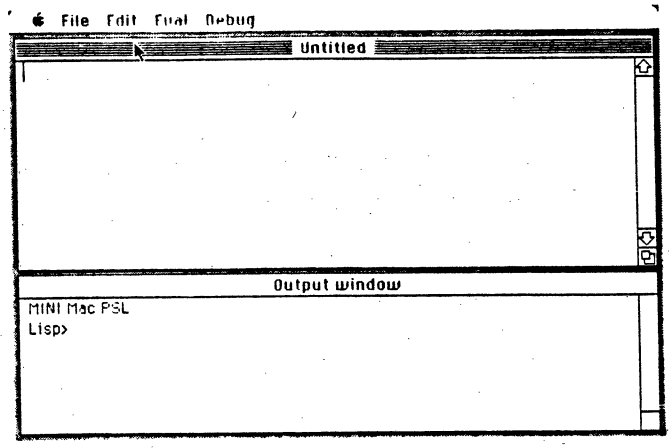
In standard Macintosh^{tm} style, to start Mac PSL, double click on the Icon (or double clicking a Mac PSL document automatically starts Mac PSL, and loads in the document).



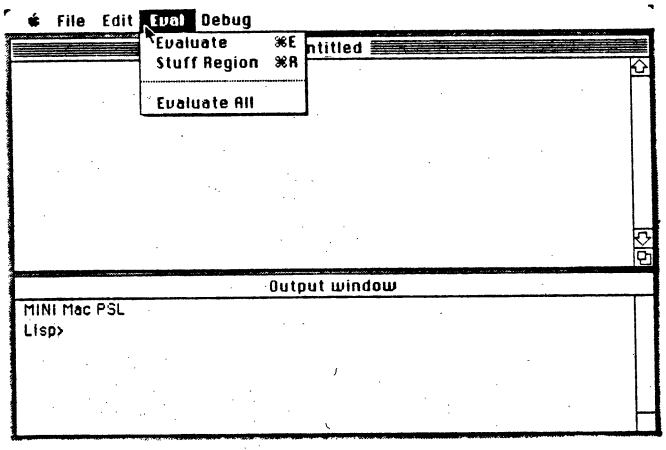
Upon startup, you are presented with two windows, one used for input, and the other used for output. The input window will be empty (unless you started Mac PSL up with a Mac

PSL document, which will be visible in the window). The output window will display a greeting message, and the "Lisp >" prompt, stating that Mac PSL is waiting for an input expression to evaluate. Also included is a set of menus:

- Apple - The standard desk accessories, including an About Mac PSL item
- File - File access operations and quitting Mac PSL
- Edit - The standard Cut, Copy and Paste operations
- Eval - Operations to submit expressions for Lisp evaluation
- Debug - Memory operation displays a dialog box that shows the amount of Macintosh memory available.



At this point, you may type PSL expressions into the input window and evaluate them. To evaluation an expression, pull down the appropriate operation under the Eval menu:

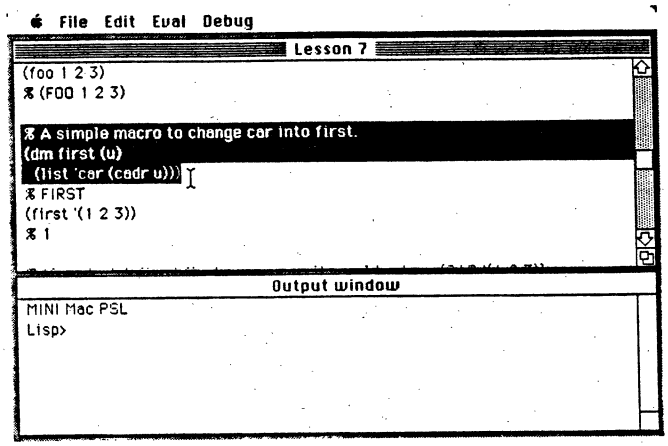


You may perform one of the following operations:

1. **Evaluate** - If the expression is a "one-liner", then use the **Evaluate** menu item (or use the *cmd-E* keystroke) to evaluate the line under the cursor.

This will extract the entire line and send it down to the output window for processing.

2. **Stuff Region** - If your expression is more than one line, then you must select the region to be evaluated (just like using cut and paste in other Mac applications) by marking the first position, and then click and drag to the end of the expression. (To mark larger regions, set the cursor at the first position, and then scroll to the end position, and shift-click). Once you have a marked region, select the **Stuff Region** command under the Eval menu (or use *cmd-R*). When evaluating many forms, Mac PSL will first echo all lines into the output window and then show the results of evaluating each form.



3. **Evaluate All** - You may also evaluate the entire input window by selecting the **Evaluate All** operation in the Eval menu.

3 Editing Lisp Code

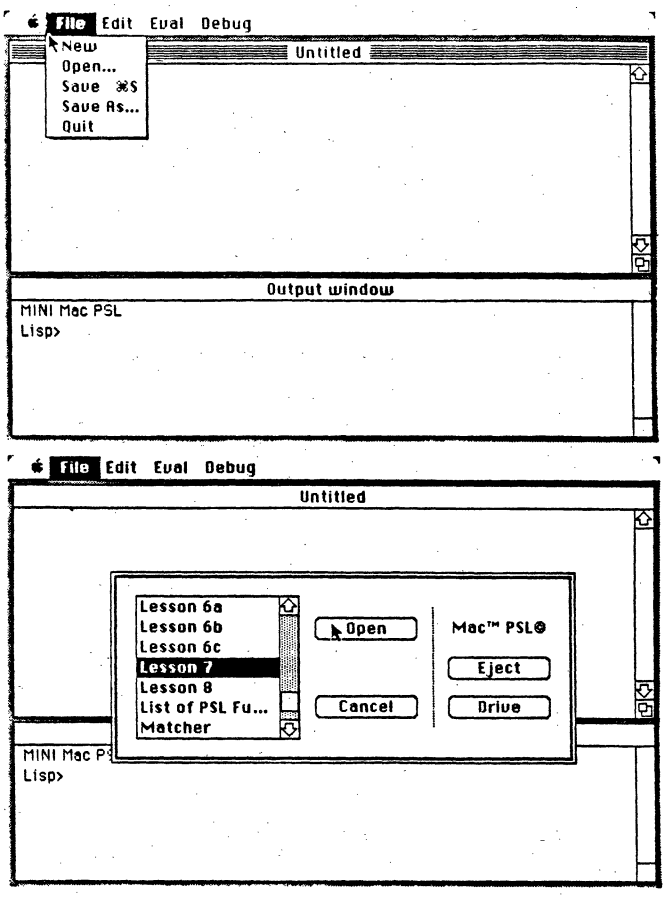
When editing Lisp code, and a right parenthesis is inserted, the cursor bounces back to the matching left parenthesis. If the matching paren is off the screen, nothing happens. Finally, if the paren is extra, Mac PSL beeps. The return key automatically indents to the first non blank character of the previous line (if you want to go to the front of the line, use the Enter key). Although it doesn't perform full auto indentation, it is at least good enough to help formatting your Lisp code.

4 Infinite Loops

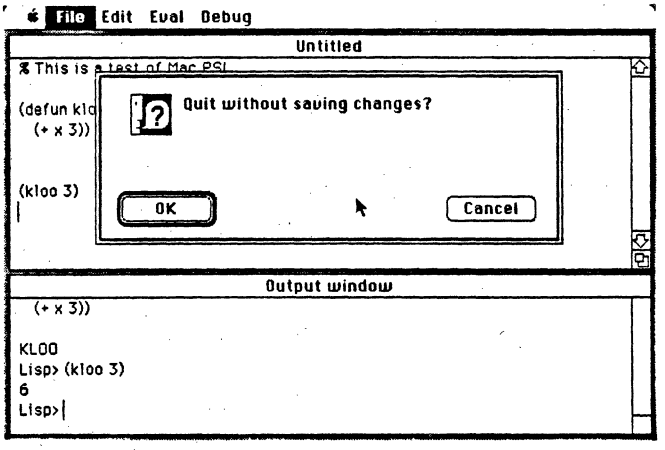
If you have a runaway computation, you may terminate it using the *Cmd-.* keystroke. This will terminate the process and return to the main read loop.

5 File Menu

The file menu provides access to external files. The user may open a file, and load it into the input window:

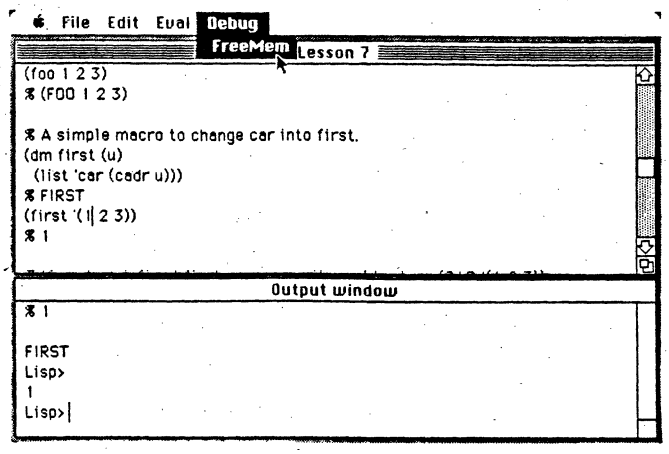


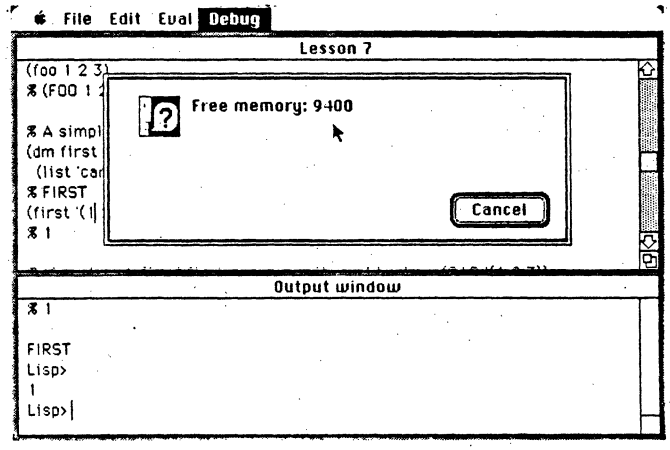
The system will give you a dialog box if you have changed the file and attempt to open a new file, clear out the input window or quit. The user may also save files and save files under a new name.



6 Debug Menu Item

When selecting this item, a dialog box will appear with the amount of Macintosh^{tm} system memory available. This has no relation to the amount of PSL pair space (which is fixed based on available memory upon startup). When this number is very small (i.e. less than 1000), items like desk accessories and large files may not be processed.





7 Garbage Collection

When running on the 128K Mac, or the 512K Mac with Ramdisk, the screen is used in the garbage collection process. When the system runs out of heap space, it will print a beginning garbage collection message. Then it will proceed to use the screen as a repository of the non-garbage Lisp items. You will see various patterns appear on the screen during this process. Once the collection is complete, the non-garbage items will be copied back to the heap, the screen will be restored and an informative message describing the number of items collected and available will be printed. *A challenge, for those of you who like challenges, is to fill the Heap with "meaningful" Lisp items that would produce some interesting pattern.*

8 File Text Editor

Included on the disk is a copy of the *File* text editor. This editor is public domain and is useful for providing a little more support than is available with Mac PSL. Its primary use is printing of files. *File* can read, edit and print any of the PSL files. It can also be used to generate a listing of an output window. To accomplish this, you have two options:

1. Paste the pertinent part of the output window into the input window and save as a file. Then with *File*, load in the file and simply print it.

2. Copy the text into the scrap, quit out of PSL, enter into *File*, and paste the scrap into a buffer. You can then print the file.

File is also useful for editing your PSL source code. However, the primary limitation is that it doesn't support paren bouncing and auto indent.

9 Known bugs for Release 1.0

1. Lots of desk accessories can cause it to bomb: Opening several desk accessories at once can cause the memory manager to panic, and the Mac will bomb with an "Out of memory" error (ID=25). *This problem appears to have been corrected for release 1.0. We have installed code to not allow a desk accessory to be opened if there is not enough room. However, some desk accessories that allocate memory after their startup (like the control panel) may still cause a crash. Therefore, be careful when using desk accessories.*

2. Stack over-flow not trapped: Writing a function that does infinite recursion will eventually bomb with a stack overflow (ID=28). *Use Cmd-. to interrupt an infinite loop before it's too late.*

3. Evaling comments is strange: The "Eval" key (*Cmd-E*) does not add a newline after comments. Thus, if a comment is first eval'ed, followed directly by an expression, the expression will be ignored. *Work-around: Use "Stuff Region" (Cmd-R) for multi-line expressions containing comments.*

4. Characters typed into the output window are processed immediately by the PSL reader as they are typed. Thus, editing keys like backspace, do not function correctly in the output window.

5. The limited number of symbols available (100) means that contrary to good programming style, usage of new Identifiers should be limited (all single character identifiers have been preallocated, so they are good candidates for names of local variables).

If you are unfamiliar with Lisp environments and editing, please open the various psl lessons supplied on the disk. Then evaluate each expression that is supplied. This should give you a good introduction to the types of facilities available with Mac PSL.